Hochschule München

Fakultät 07 Informatik und Mathematik

Datenzugriffskomponenten

mit

JPA 2.1

(Grundlagen der Java Persistence Architecture)

FWP Fach:	**Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen**
Bearbeitet von:	Vladislav Faerman, 02929612
Betreuer:	Herr Michael Theis
Abgabedatum:	15.05.2015

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Seminararbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15.05.2015

Inhaltsverzeichnis

I. Einleitung

In dieser Arbeit wird gezeigt, wie Java Persistence API (JPA) die Verwaltung der Klasseninstanzen der objektorientierten Programmiersprache Java in den relationalen Datenbanken ermöglicht.

JPA ist ein Standard für das Objekt-Relationales Mapping (ORM). ORM ist ein Verfahren zur Speicherung von Objekten in Datenbanken. Die Klassen aus der objektorientierten Welt werden auf Tabellen abgebildet und die Objekte der Klassen werden in diesen Tabellen gespeichert.

JPA ist ein Standard, der in einem Java Community Process entworfen wird. Die aktuelle Version ist JPA 2.1[1]. Es gibt verschiedene Implementierungen von JPA. Die aktuelle Referenzimplementierung für JPA 2.1. ist EclipseLink[2].

II. Konfiguration

1. Einbindung von JPA in Java App

Um die Verwendung von JPA in einer Java Anwendung zu ermöglichen, müssen die entsprechenden Bibliotheken dem Java Projekt hinzugefügt werden. Die **Abbildung *1*** zeigt eine Maven[3] Konfigurationsdateidatei. Maven wird verwendet, um die Abhängigkeiten zu den externen Bibliotheken zu verwalten. Die benötigten Bibliotheken werden von dem User in der Maven- Konfigurationsdatei pom.xml als Abhängigkeiten (engl. dependencies) deklariert und das Tool lädt die Bibliotheken automatisch herunter, speichert die in einem lokalen Repository und fügt sie dann dem Projekt hinzu.

---

[1] Link: http://en.wikibooks.org/wiki/Java_Persistence, abgerufen am 16.04.2015

[2] Link: http://eclipse.org/eclipselink/#jpa, abgerufen am 16.04.2015

[3] Link: https://maven.apache.org, abgerufen am 16.04.2015

In dem Beispiel, in der **Abbildung *1*** sind zwei dependencies zu sehen. Die Hibernate-Entitymanager Bibliothek enthält die JPA Interfaces und die JPA Implementierung von Hibernate[4]. Die JPA kommuniziert mit der Datenbank über eine JDBC[5]-Schnittstelle, deswegen wird ein Treiber für die entsprechende Datenbank benötigt. In dem Beispiel in der **Abbildung *1*** wird der Treiber für die MySQL- Datenbank deklariert.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>edu.hm.maven</groupId>
	<artifactId>firstAppWithJPA</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<dependencies>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-entitymanager</artifactId>
			<version>4.3.6.Final</version>
		</dependency>
		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<version>5.1.31</version>
		</dependency>
	</dependencies>
</project>
```

Abbildung 1 Wichtige Bibliotheken für die JPA- Verwendung

2. Verbindung zur Datenbank

Für die Kommunikation zur Datenbank muss die Verbindung bekannt gemacht werden. JPA erwartet eine XML Datei, die die Verbindungen zu den Datenbanken konfiguriert: src/main/resources/META-INF/persistence.xml. Die **Abbildung *2*** zeigt ein Beispiel für die Datei persistence.xml.

[4] Link: http://hibernate.org, abgerufen am 16.04.2015

[5] Link: http://www.oracle.com/technetwork/java/javase/jdbc/index.html, abgerufen am 16.04.2015

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
version="2.1">
  <persistence-unit name="jpa-example" transaction-type="RESOURCE_LOCAL">
     <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
       <properties>
         <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/MySQLForJPA" />
         <property name="javax.persistence.jdbc.user" value="friendOfJPA" />
         <property name="javax.persistence.jdbc.password" value="password" />
         <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />

         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.format_sql" value="true" />

         <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
         <property name="hibernate.hbm2ddl.auto" value="create" />


         <!-- Configuring Connection Pool -->
         <property name="hibernate.c3p0.min_size" value="5" />
         <property name="hibernate.c3p0.max_size" value="20" />
         <property name="hibernate.c3p0.timeout" value="500" />
         <property name="hibernate.c3p0.max_statements" value="50" />
         <property name="hibernate.c3p0.idle_test_period" value="2000" />
       </properties>
  </persistence-unit>
</persistence>
```

A

B

Abbildung 2 Konfigurieren der persistence.xml

In der **Abbildung *2*** wird eine „persistence-unit“ namens „jpa-example“ deklariert, die die Verbindung zu einer konkreten Datenbank beschreibt.

Die Properties in der **Abbildung *2***, die mit **A** beschriften sind, sind von JPA vorgegeben und definieren die Verbindungsinformationen.

Die Hibernate- Implementierung bringt zusätzliche Tools mit, die die Arbeit mit JPA vereinfachen. Die mit **B** beschriftete Property „hibernate.hbm2ddl.auto“ definiert was mit Datenbankschema beim Start der Anwendung passiert. Mögliche Werte für das Property sind validate, update, create und create-drop. Wie die verschiedenen Werte beim Starten der Anwendung auf das Datenbankschemata auswirken, ist in der **Tabelle *1*** beschrieben.

Wert	Auswirkung
validate	Validierung des Schemas, aber keine Veränderung an der Datenbank. Wenn das Datenbankschema zu dem aktuellen Mapping nicht kompatibel ist, wird eine Fehlermeldung produziert.

update	Falls das Datenbankschema zu dem aktuellen Mapping nicht kompatibel ist, werden die betroffenen Tabellen angepasst. Die Daten in den veränderten Tabellen können dadurch verloren gehen.
create	Beim Start der Anwendung werden alle Tabellen in der Datenbank gelöscht und das aktuelle Schema neuangelegt. Wenn die Anwendung herunterfährt, bleibt das Datenbankschema in der Datenbank.
create-drop	Beim Start der Anwendung werden alle Tabellen in der Datenbank gelöscht und das aktuelle Schema neuangelegt. Wenn die Anwendung herunterfährt, werden alle Tabellen in der Datenbank gelöscht.

Tabelle 1 Mögliche Werte für das Property „hibernate.hbm2ddl.auto“

III. Objekt-Relationales Mapping (ORM) mit JPA

1. Entitäten und die Beziehungen zwischen Entitäten

Eine Entität ist ein eindeutig identifizierbares Objekt des Datenmodells. Diese wird durch Eigenschaften beschrieben und es lässt sich durch diese Eigenschaften von anderen Objekten unterscheiden.

In dem Datenmodell werden Entitäten und Beziehungen (Relationen) zwischen Entitäten beschrieben. Es gibt insgesamt vier Beziehungstypen:

a) Eins-zu-eins-Beziehung (1:1)

b) Eins-zu-viele-Beziehung (1:n)

c) Viele-zu-eins-Beziehung (n:1)

d) Viele-zu-viele-Beziehung (m:n)

Die Beziehungen können bi- und unidirektional sein:

- die Beziehung ist unidirektional, wenn die Verbindung nur von einem der beteiligten Entitäten deklariert wird,
- die Beziehung ist bidirektional, wenn die Verbindung von beiden beteiligten Entitäten deklariert wird.

2. Mapping einer Klasse

Damit die Objekte einer Java Klasse von JPA verwaltet werden, muss die Annotation @Entity über den Klassennamen platziert werden. Dadurch wird die Klasse zu einer Tabelle in der Datenbank gemappt. Der Name der entsprechenden Tabelle kann mit dem Attribut name der @Entity Annotation konfiguriert werden.

Die primitiven Attribute der gemappten Klasse werden in der relationalen Welt als Spalten der entsprechenden Tabelle abgebildet. Wenn bestimmte Attribute nicht persistent abgespeichert werden sollten, muss die Annotation @Transient über dem bestimmten Attribut stehen.

Ein Klassenattribut, das JPA als Primärschlüssel, verwenden soll ist mit @Id zu annotieren. JPA bietet auch die Möglichkeit der Primärschlüssel zu generieren. Die Attribute muss dafür zusätzlich zu @Id mit @GeneratedValue annotiert werden.

Wenn der Primärschlüssel aus mehr als einem Attribut zusammengesetzt ist, muss die Klassenstruktur verändert werden. Die zu Schlüssel gehörenden Attribute werden in die eigene interne Klasse ausgelagert. Diese Klasse wird mit @Embeddable annotiert. Das Objekt der Elternklasse bekommt eine Referenz zu einem Objekt der internen Klasse, die mit @EmbeddedId annotiert wird.

Die bereits beschriebene Annotationen sowie alle weiteren JPA-Annotationen, auf die später eingegangen wird, sind im Package javax.persistence.* definiert.

3. Unidirektionale 1:1- Beziehung

Um eine 1:1- Beziehung mit JPA zu realisieren, wird die Annotation @OneToOne verwendet. Die Annotation wird über dem Klassenattribut platziert, der die Referenz zu einer anderen Klasse präsentiert. Die referenzierte Klasse muss auch von JPA verwaltet werden.

Zur Realisierung der 1:1- Beziehung zwischen zwei Entitäten in der relationalen Datenbank ergänzt JPA die zu einer der Entitäten gemappte Tabelle mit zusätzlichen Spalten. Diese Spalten bilden den Fremdschlüssel auf den primären

Schlüssel der Tabelle, zu der die andere Entität gemappt ist. Der Name der Fremdschlüsselspalte kann mit @JoinColumn konfiguriert werden.

In der **Abbildung *3*** sind zwei Klassen Auto und Autobesitzer zu sehen. Es besteht eine 1:1- Verbindung zwischen den beiden Klassen. Die Beziehung ist unidirektional: die Klasse Autobesitzer wird aus der Klasse Auto referenziert, die Klasse Autobesitzer enthält aber keine Referenz zu der Klasse Auto.

```java
@Entity
public class Auto {

	@Id @GeneratedValue
	private long id;

	private String marke;
	private String modell;
	private String kennzeichen;

	@OneToOne
	private Autobesitzer autoBesitzer;

	// Getter und Setter
}
```

```java
@Entity
public class Autobesitzer {

	@Id @GeneratedValue
	private long id;

	private String vorname;
	private String nachname;

	// Getter und Setter
}
```

Abbildung 3 Unidirektionale1:1- Beziehung Auto – Autobesitzer

In dem Beispiel in der **Abbildung *3*** wird die Referenz zu der Klasse Autobesitzer mit @OneToOne annotiert. Dadurch wird die Tabelle der entsprechenden Klasse Auto mit zusätzlicher Spalte erweitert, die einen Fremdschlüssel zu der Tabelle Autobesitzer repräsentiert. In der **Abbildung 4** ist zu sehen wie der Name der Spalte mit @JoinColumn- Annotation konfiguriert wird.

```java
@OneToOne
@JoinColumn(name = "Autobesitzer")
private Autobesitzer autoBesitzer;
```

Abbildung 4 Weitere JPA- Annotation @JoinColumn

Die endgültige Tabellenstruktur mit dem Fremdschlüssel zur Tabelle Autobesitzer ist in der **Abbildung 5** zu sehen.

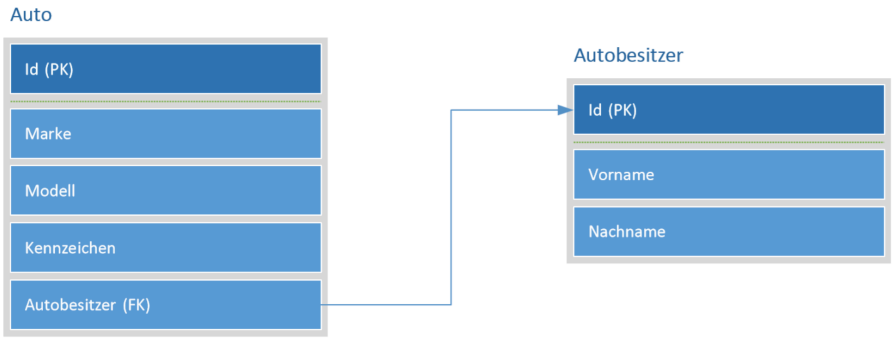

Abbildung 5 Tabellenstruktur für die Beziehung Auto – Autobesitzer

Wenn die Annotation @JoinColumn nicht angewendet wird, wird der Spaltenname folgendermaßen gebildet: Name der Zielklasse, in unserem Fall Autobesitzer, wird mit dem Unterstrich und dem Namen des Primärschlüssels der Zielklasse, in unserem Fall id, aneinandergesetzt. Als Ergebnis würde sich somit folgender Spaltenname: autoBesitzer_id ergeben.

4. Bidirektionale 1:1- Beziehung

Das vorherige Beispiel mit der Beziehung zwischen dem Auto und Autobesitzer ist unidirektional. Um die 1:1- Beziehung bidirektional zu modellieren, ist es notwendig die Beziehung in den beiden Entitäten zu deklarieren. Damit JPA erkennt, dass es sich tatsächlich um eine bidirektionale Beziehung handelt und nicht um zwei unabhängige unidirektionale Beziehungen in beide Seiten, muss es explizit definiert werden. Die mappedBy Attribute der @OneToOne- Annotation gibt an, dass die Verbindung an einer anderen Stelle bereits deklariert wurde und wiederverwendet werden sollte. Der Wert für mappedBy ist das Klassenattribut in der referenzierten Klasse, das die Verbindung realisiert.

In der **Abbildung 6** sind sowohl im Auto als auch in dem Autobesitzer das entsprechende Property mit @OneToOne annotiert. Für die @OneToOne in Autobesitzer Klasse ist angegeben, dass die Verbindung bereits auf der anderen Seite existiert und wiederverwendet werden muss. Als Wert ist "autoBesitzer" angegeben - der Name des Attributs, das diese Verbindung in der Klasse Auto realisiert.

```java
public class Auto {

	@Id @GeneratedValue
	private long id;

	private String marke;
	private String modell;
	private String kennzeichen;

	@OneToOne
	@JoinColumn(name = "Autobesitzer")
	private Autobesitzer autoBesitzer;

	// Getter und Setter
}
```

```java
@Entity
public class Autobesitzer {

	@Id @GeneratedValue
	private long id;

	private String vorname;
	private String nachname;

	@OneToOne(mappedBy = "autoBesitzer")
	private Auto auto;

	// Getter und Setter
}
```

Abbildung 6 Bidirektionale1:1- Beziehung Auto – Autobesitzer

Die Änderungen, die vorgenommen werden um aus einer unidirektionalen Beziehung eine bidirektionale 1:1 Beziehung zu machen, haben keine Auswirkung auf die schon existierende Tabellenstruktur. Die Tabellenstruktur für die Klassen in ***Abbildung 6*** ist auch in der **Abbildung 5** abgebildet.

5. 1:n- Beziehung

Die @OneToMany oder @ManyToOne Annotationen werden verwendet, um eine 1:n-Beziehung mit JPA zu realisieren.

Das Klassenattribut auf der 1 Seite, das die Menge der Referenzen zu der anderen Klasse enthält, wird mit @OneToMany annotiert. Für den Typ des Klassenattributs sind die Klassen, die Collection, Set, List oder Map aus dem Package java.util implementieren erlaubt.[6]

Auf der n- Seite wird die entsprechende Referenz mit @ManyToOne annotiert.

Die bidirektionale Verbindung wird mittels mappedBy Attributs der @OneToMany Annotation realisiert.

In der **Abbildung 7** ist ein Beispiel für eine bidirektionale 1:n- Beziehung präsentiert. Eine Firma beschäftigt mehrere Mitarbeiter und jeder Mitarbeiter ist nur bei einer Firma tätig.

[6] Vgl. Bernd Müller, Harald Wehr, Java Persistence API 2 Hibernate, EclipseLink, OpenJPA und Erweiterungen, 2012, Kapitel 4.1 Objekte und Beziehungen, Seite 96

```java
@Entity
public class Firma {

	@Id @GeneratedValue
	private long id;

	private String firmenName;

	@OneToMany(mappedBy = "firma")
	@JoinColumn(name = "FirmenID")
	private List<Mitarbeiter> mitarbeiters;

	// Getter und Setter
}
```

```java
@Entity
public class Mitarbeiter {

	@Id @GeneratedValue
	private long id;

	private String vorname;
	private String nachname;
	private String position;

	@ManyToOne
	private Firma firma;

	// Getter und Setter
}
```

Abbildung 7 Bidirektionale 1:n- Beziehung Firma – Mitarbeiter

In der Datenbank wird die 1:n- Relation durch eine Fremdschlüsselbeziehung realisiert. Die Tabelle, die Entität auf der n- Seite repräsentiert, referenziert die Tabelle auf der n- Seite (siehe **Abbildung 8**).

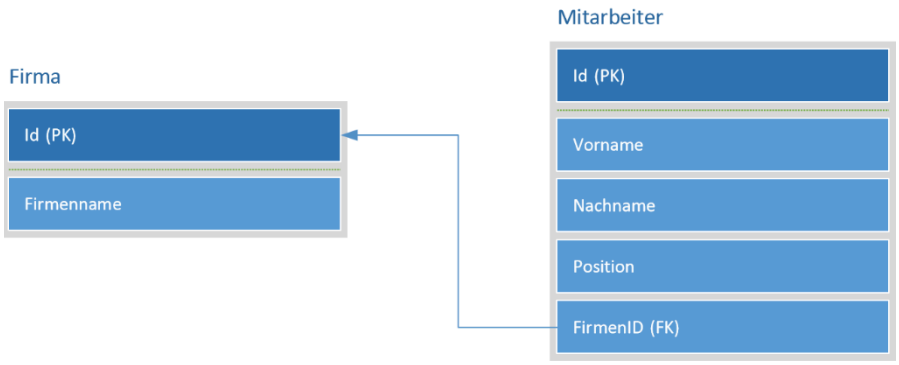

Abbildung 8 Tabellenstruktur für die Beziehung Firma – Mitarbeiter

6. n:m- Beziehung

Eine n:m- Beziehung wird in JPA mit der @ManyToMany- Annotation definiert. Für eine unidirektionale n:m- Beziehung muss nur die besitzende Seite die @ManyToMany- Annotation enthalten. Für eine bidirektionale Beziehung wird @ManyToMany auf beiden Seiten benutzt und mappedBy Attribut bei einer der Annotationen verwendet. In der Datenbank wird von JPA für eine n:m- Beziehung eine sogenannte Join-Tabelle angelegt. Join- Tabelle ist eine separate Tabelle, die nur aus Fremdschlüsseln besteht. Die Fremdschlüssel referenzieren die beiden Tabellen, auf die die Klassen abgebildet werden.

Die JPA- Spezifikation bietet die @JoinTable- Annotation an, um den Namen der Join Tabelle definieren zu können. Wenn @JoinTable nicht verwendet wird, wird der Name der Tabelle folgendermaßen definiert: Entität-Name des Besitzers der Beziehung dann Unterstrich und anschließend Entität-Name der anderen Seite.

Anhand eines Beispiels in der **Abbildung 9** wird die Verwendung der n:m- Beziehung demonstriert. Ein Dozent betreut mehrere Studenten und jeder Student kann von mehreren Dozenten betreut werden.

```java
@Entity
public class Student {

	@Id @GeneratedValue
	private int id;
	private String name;

	@ManyToMany
	@JoinColumn(name = "Dozent")
	private List<Dozent> dozents;

	// Getter und Setter
}
```

```java
@Entity
public class Dozent {

	@Id @GeneratedValue
	private int id;
	private String name;

	// Getter und Setter
}
```

Abbildung 9 Unidirektionale n:m- Beziehung Student – Dozent

Die Tabellenstruktur für das Beispiel ist in der **Abbildung 10** gezeigt.

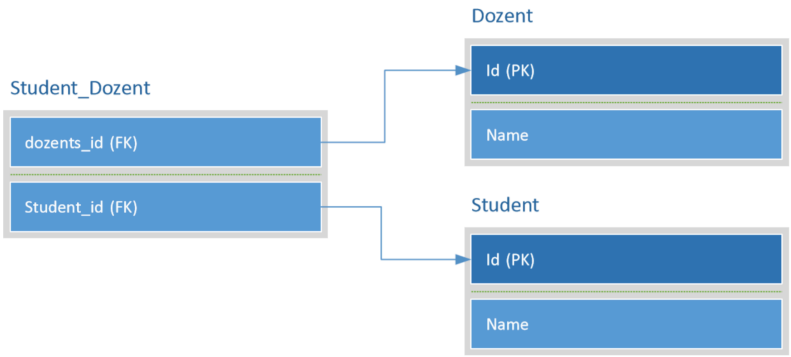

Abbildung 10 Tabellenstruktur für die Beziehung Dozent – Student

7. Vererbung

Dieses Kapitel beschreibt verschiedene Möglichkeiten, die JPA zur Realisierung von Vererbungsbeziehungen bereitstellt. In der JPA 2.1 sind drei Vererbungsstrategien vorgesehen:

1. SINGLE_TABLE
2. JOINED
3. TABLE_PER_CLASS

Die Umsetzung der Vererbungsbeziehungen erfolgt in JPA mit der @Inheritance-Annotation und Setzen des Attributs strategy auf eine von drei angebotenen Alternativen, SINGLE_TABLE, JOINED oder TABLE_PER_CLASS. Die TABLE_PER_CLASS ist die Default- Strategie für die Implementierung der Vererbung.

Wie die verschiedenen Vererbungsstrategien konfiguriert werden und wie die in der Datenbank abgebildet werden, wird anhand eines Beispiels erläutert. Die Klassen Student und Dozent erweitern eine abstrakte Oberklasse Person. Die **Abbildung 11** zeigt das entsprechende UML-Klassenmodell.

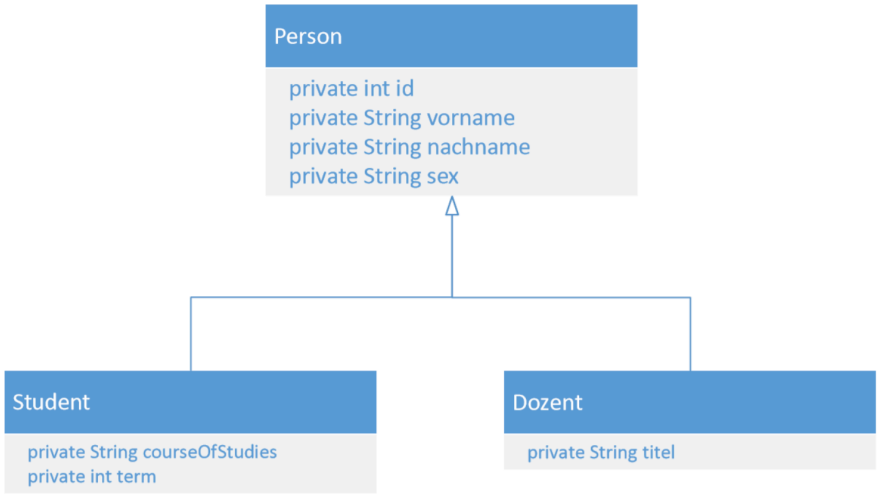

Abbildung 11 Allgemeine Vererbungshierarchie der Kundenklassen

1) SINGLE_TABLE

Wie der Name schon sagt, bei SINGLE_TABLE- Strategie ist eine einzige Tabelle für die gesamte Vererbungshierarchie verantwortlich. Die Attribute aller Klassen werden auf Spalten in einer gemeinsamen Datenbanktabelle abgebildet. Je nach gespeichertem Objekt bleiben für den jeweiligen Typ nicht benötigte Datenbankspalten leer. Die Klassenzugehörigkeit des Objekts wird in einer Diskriminationsspalte gespeichert.

Der Name der Diskriminationsspalte und die Werte für verschiedene Klassen können konfiguriert werden, um die Übersichtlichkeit zu verbessern. Der Name der Diskriminationsspalte wird mit @DiscriminatorColumn- Annotation bei der Elternklasse definiert. Die Werte für verschiedene Klassen werden mit @DiscriminatorValue bei den möglichen Klassen definiert, der Default Wert ist der Klassenname. Die Werte können Strings, chars und integers sein.

Die Konfiguration der SINGLE_TABLE- Strategie für unser Beispiel ist in **Abbildung*en* *14*** und ***15*** zu sehen. Der Name der Diskriminationsspalte ist bei der Elternklasse Person definiert. Da die Elternklasse abstract ist, können nur Objekte der Klassen Student und Dozent existieren, deswegen ist @DiscriminatorValue- Annotation nur bei diesen Klassen sinnvoll.

```
@Entity
@DiscriminatorColumn(name = "PersKlassen")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Person {

	@Id @GeneratedValue
	private long id;
	private String vorname;
	private String nachname;
	private String sex;

	// Getter und Setter
}
```

Abbildung 12 Vererbung als Default-Wert SINGLE_TABLE

```
@Entity
@DiscriminatorValue("Student")
public class Student extends Person {

	private String courseOfStudies;
	private int term;

	@ManyToMany
	private List<Dozent> dozents;

	 // Getter und Setter
}
```

```
@Entity
@DiscriminatorValue("Dozent")
public class Dozent extends Person {

	private String titel;

	@ManyToMany(mappedBy = "dozents")
	private List<Student> students;

	 // Getter und Setter
}
```

Abbildung 13 Unterklassen Student und Dozent

Wie die Tabelle für die Klassen und Konfiguration in den ***Abbildungen 14*** und ***15*** aussieht, ist in der ***Abbildung 13*** zu sehen.

PersKlassen	id	nachname	sex	vorname	titel	courseOfStudies	term
Dozent	1	Theis	männlich	Michael	Dozent	NULL	NULL
Student	2	Mustermann	männlich	Markus	NULL	Informatik	3

Abbildung 13 Dozent und Student mit SINGLE_TABLE

2) JOINED

Bei der JOINED- Strategie ist für jede Klasse in der Hierarchie eine Tabelle vorgesehen. Jede Tabelle enthält die Spalten nur für die Attribute, die nur direkt in der gemappten Klasse deklariert sind. Um die Objekte der abgeleiteten Klassen abzufragen ist join mit der Elterntabelle notwendig.

Die Klassenstruktur für unser Beispiel für die JOINED- Strategie ist in der **Abbildung 14** zu sehen.

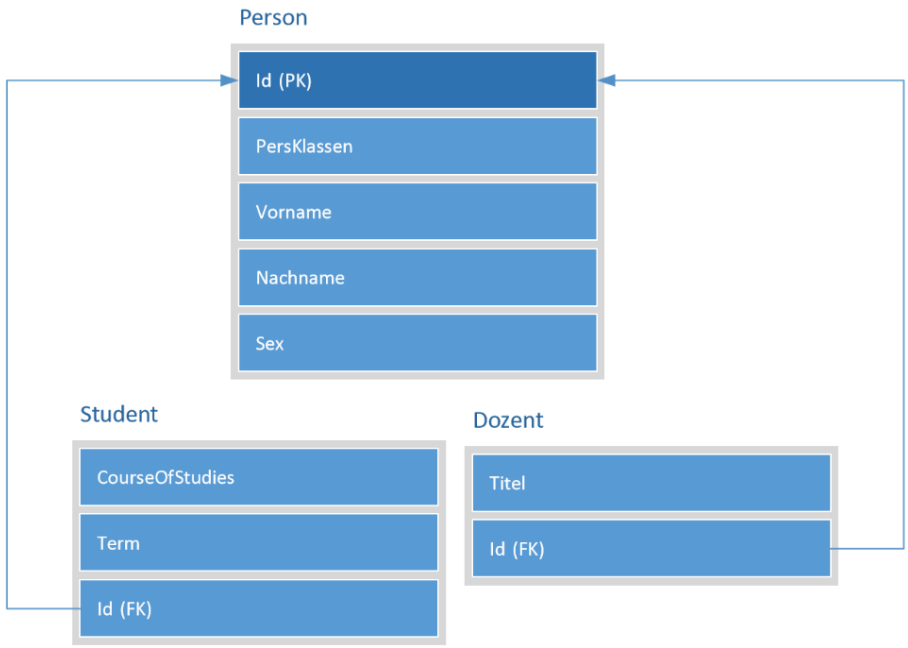

Abbildung 14 Bildung der Tabellenstruktur nach JOINED- Strategie

3) TABLE_PER_CLASS

Für TABLE_PER_CLASS- Strategie wird für jede nicht abstrakte Klasse eine eigene Tabelle angelegt. Jede Tabelle enthält die Spalten für die Attribute, die in der gemappten Klasse und in den Eltern der gemappten Klasse deklariert sind.

Somit sind alle die Objekte einer bestimmten Klasse in einer Tabelle gespeichert. Für die polymorphen Abfragen (wenn die Objekte der Unterklasse abgefragt werden und die Objekte aller abgeleiteten Klassen erwartet werden) ist eine UNION- Operation notwendig. Die Klassenstruktur für unser Beispiel für die TABLE_PER_CLASS-Strategie ist in der ***Abbildung 17*** zu sehen.

In der JPA- Spezifikation ist die Strategie optional und wird nicht von allen Providern unterstützt.

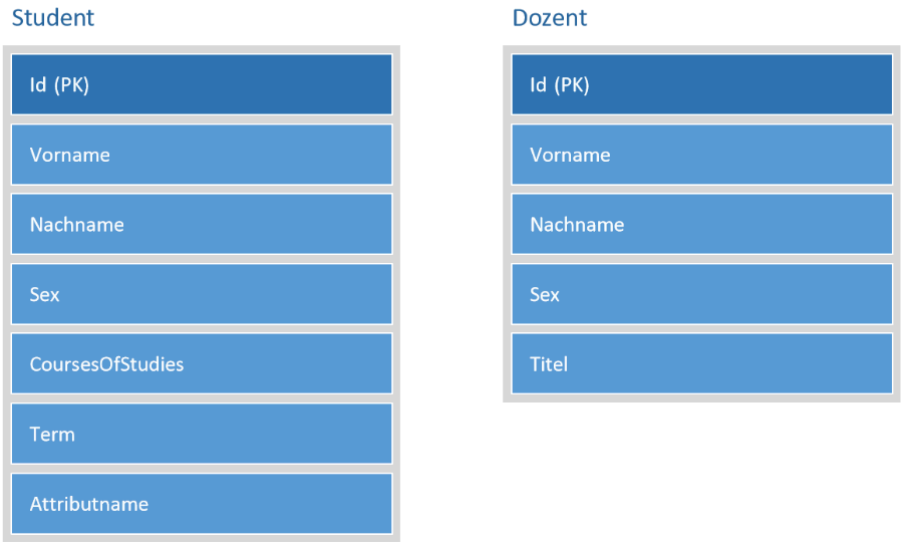

Abbildung 15 Bildung der Tabellenstruktur nach TABLE_PER_CLASS- Strategie

IV. Das zentrale Konzept der JPA

1. EntityManager

javax.persistence.EntityManager ist die zentrale Komponente in JPA. EntityManager verwaltet die Schnittstelle zu der Datenbank und alle Operationen werden mit den Entitäten von dem EntityManager ausgeführt (konkreter von der Klasse, die EntityManager impelementiert). Die wichtigsten Methoden von EntityManager Interface sind in der ***Tabelle 2*** beschrieben.

Methoden	Beschreibung
persist()	Speichern eines Objektes in der Datenbank
remove()	Entfernen eines Objekts aus der Datenbank
find()	Finden eines Objekts aus der Datenbank über den Primärschlüssel (PK)
refresh()	Aktualisieren eines Objekts aus der Datenbank
createQuery()	Erzeugen einer Query mit JPQL
getTransaction()	gibt die aktuelle Transaktion zurück.

Tabelle 2 Liste der wichtigsten Methoden

Eine Transaktion ist eine logische Einheit von Operationen innerhalb der Datenbank. Die Transaktion kann vom EntityManager explizit gestartet, committed werden oder der EntityManager startet für jede Operation eine eigene Transaktion, siehe **Tabelle *3***.

```
EntityTransaction etrans = em.getTransaction();
```

Tabelle 3 Instanziierung einer Transaktion

```
em.getTransaction().begin();
```

Tabelle 4 Ohne Instanziierung einer Transaktion

Die explizit von User gesteuerte Transaktion (vgl. etrans **Tabelle *3***) muss mit der Methode begin() beginnen und mit commit() abgeschlossen werden. Innerhalb der laufenden Transaktion können die Daten verändert werden. Zum Beispiel mit der Methode persist() werden die Objekte der Entitäten-Klassen in der Datenbank abgespeichert.

2. Kaskadierung

Für die Assoziationsannotationen, die in dem Kapitel 4 beschrieben wurden, können die Operationen definiert werden, die auch für die Instanzen ausgeführt werden, mit denen die Beziehung besteht. Dies erfolgt durch Attribut cascade, das einen Enum Wert aus javax.persistence.CascadeType erwartet. Die möglichen Werte und die Auswirkung auf die referenzierten Entitäten sind in der ***Tabelle 7*** beschrieben. Wenn die Kaskadierung nicht definiert ist, werden keine Operationen kaskadiert. Die Assoziationsannotationen wie @OneToOne, @OneToMany und @ManyToMany sind

wie in der **Tabelle *5*** mit dem Attribut cascade zu erweitern. Die folgende Liste zeigt an, welche Kaskadierungstypen zur Verfügung stehen:

CascadeType[7]	Beschreibung
PERSIST	Beim Persistieren einer Entität werden referenzierte Entitäten gespeichert.
REMOVE	Beim Löschen einer Entität werden auch referenzierte Entitäten gelöscht.
REFRESH	Beim Aktualisieren einer Entität werden die referenzierten Entitäten aus der Datenbank erneut geladen.
MERGE	Bei der Merge- Operation werden die Änderungen an einer Entität in die Datenbank gespeichert. Für die referenzierten Entitäten bedeutet es, dass Ihre Änderungen auch in die Datenbank geschrieben werden.
ALL	Alle beschriebene Operationen werden auch für die referenzierte Entitäten durchgeführt.

Tabelle 5 Die Liste der Kaskadierungstypen

In dem Beispiel in der **Tabelle *6*** ist für die 1:1 Beziehung die CascadeType.ALL definiert. Das hat zur Folge, dass alle Operationen, die mit Auto Objekt durchgeführt werden, auch mit dem referenzierten Autobesitzer Objekt durchgeführt werden. Wenn Auto z. B aus der Datenbank gelöscht wird, wird sein Besitzer auch gelöscht.

Andernfalls wird das Auto des Autobesitzers nicht automatisch mitgelöscht, solange in der Klasse Autobesitzer für das auto- Attribut keine Kaskadierung definiert ist.

Entität-Klasse: Auto

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "Autobesitzer")
private Autobesitzer autoBesitzer;
```

Entität-Klasse: Autobesitzer

```
@OneToOne(mappedBy = "autoBesitzer")
@JoinColumn(name = "Auto")
private Auto auto;
```

Tabelle 6 Erweiterung der @OneToOne- Annotation mit CascadeType.ALL

[7] Link: http://ci.apache.org/projects/openjpa/trunk/docbook/manual/jpa_overview_meta_field.html#jpa_overview_meta_cascade, abgerufen am 01.05.2015

3. JPQL (Java Persistence Query Language – SQL unter JPA)

JPQL ist die Abfragesprache für die Entitäten, die von JPA verwaltet werden. Die Abfragesprache ist an SQL angelehnt, operiert mit Klassen und Klassenattributen. Die JPQL- Abfragen werden von JPA in SQL übersetzt und die Operationen werden in der Datenbank ausgeführt.

Ein Beispiel für die Implementierung einer solchen JPQL-Abfrage ist in der **Tabelle 7** abgebildet.

```
Query query = em.createQuery("select b from Autobesitzer b");
query.getResultList();
```

Tabelle 7 Typische JPQL- Abfrage

Eine Instanz der Klasse Query erhält man mit einer Methode der bereits bekannten EntityManagers (em):

- createQuery()

Neu hinzugekommen in JPA 2.0 ist das Interface TypedQuery. Dieses Interface erbt vom bereits vorgestellten Interface Query. Entscheidender Vorteil dieses Interfaces ist der Umstand, dass im Ergebnis der Abfrage bereits typisierte Objekte zurückgegeben werden.[8]

Die Methode getResultList() liefert eine Liste vom Typ java.util.List. Welche Typen diese Liste welcher Stelle beinhaltet, bestimmt der Ausdruck hinter der Select-Anweisung. Die Methode getResultList() des Interfaces TypedQuery liefert bereits eine vollständige typisierte Liste von Objekten zurück.[9]

Es werden die SELECT, UPDATE und DELETE Operationen unterstützt.

Die Abfragen werden anhand von Beispielen erläutert.

---

[8] Vgl. Bernd Müller, Harald Wehr, Java Persistence API 2 Hibernate, EclipseLink, OpenJPA und Erweiterungen, 2012, Kapitel 7.1 JPA-Query-Interfaces, Seite 178

[9] Vgl. Bernd Müller, Harald Wehr, Java Persistence API 2 Hibernate, EclipseLink, OpenJPA und Erweiterungen, 2012, Kapitel 7.1 JPA-Query-Interfaces, Seite 179

3.1. Select- Operation

a. select b from Autobesitzer b

b. select b from Autobesitzer b where b.vorname='Markus' and b.nachname='Mustermann'

c. select distinct b from Autobesitzer b where b.nachname in ('Faerman')

Es können auch nur bestimmte Attribute abgefragt werden:

d. select a.marke from Auto a

3.2. Join- Operation

e. select s from Student s, Dozent d where s.nachname=d.nachname

f. select a from Auto inner join a.autoBesitzer b where b.vorname='Markus' and b.nachname='Mustermann'

Es wird auch Left Outer Join Operation unterstützt.

3.3. Fetch join- Operation

Fetch join kann nur mit Entitäten verwendet werden, mit welchen die Relation deklariert wird. Bei der Verwendung von Fetch join werden die entsprechende referenziert Objekte gleich mitgeladen. Fetch join wird für alle join typen unterstützt:

g. select s from Student s fetch inner join dozents d

3.4. Group by

h. select s, count(d) from Student s inner join dozents d group by(s)

3.5. Update- Operation

i. update Autobesitzer b set b.nachname=’Theis’ where b.nachname=’Mustermann’

3.6. Delete- Operation

j. delete from Autobesitzer b where b.nachname='Faerman'

V. Zusammenfassung und Fazit

Die relationalen Datenbanken haben sich im Laufe der Zeit gut bewährt für die Verwaltung der persistenten Daten. Die Persistenz der Daten aus einer Anwendung in einer relationalen Datenbank ist mit viel Aufwand verbunden. Die entsprechenden Tabellenstrukturen müssen entworfen und zu den Strukturen in der Anwendung gemappt werden. Das Persistieren und Abfragen mit dem manuellen Generieren von SQL- Abfragen ist mühsam und fehleranfällig, besonders wenn die Applikation verändert wird und die Datenstrukturen erweitert oder geändert werden.

JPA ist ein Werkzeug, das bei all diesen Problemen helfen sollte. Die Idee dabei ist, dass man sich auf die Anwendung konzentriert und das Persistieren in der Datenbank von JPA übernommen wird. Meiner Meinung nach ist es gut gelungen. Man muss sich zwar weiterhin Gedanken darüber machen, wie die Daten in der Datenbank gespeichert werden, um die optimale Performance zu erzielen. Gleichzeitig ist man aber von vielen Routineaufgaben befreit. Man kann auf die Best Practices für das Mapping von der Objektorientierten zu Relationalen Welten zurückgreifen, die JPA out-of-the box bietet. Durch die kluge Konfiguration z. B. mit Kaskadierung wird einem auch viel Arbeit abgenommen.

VI. Verzeichnis und Implementierung

1. Literaturverzeichnis

Linda DeMichiel: Java Persistence API, Version 2.1, URL: http://download.oracle.com/otn-pub/jcp/persistence-2_1-fr-eval-spec/JavaPersistence.pdf?AuthParam=1432068991_e315bb268c833c96b87a33a56608e0e5, Zugriff: 15.04.2015

Bernd Müller, Harald Wehr: Java Persistence API 2 Hibernate, EclipseLink, OpenJPA und Erweiterungen, Carl Hanser Verlag München, 2012

Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, Kim Haase: The Java EE5 Tutorial, URL: http://docs.oracle.com/javaee/5/tutorial/doc/?wp406143&PersistenceIntro.html#wp78460, Zugriff 17.04.2015

2. Eigene Beispielimplementierung

https://github.com/vlxxxfa/JPAExample.git